COMMAND-LINE TEXT PROCESSING SERIES

AWK

Fields, patterns, arrays, control flow, and real-world one-liners — the companion course to Regex and SED. 8 complete chapters.

8 Chapters

osztromok.com

CHAPTER 1: INTRODUCTION TO AWK

CHAPTER 1

Introduction to AWK

What it's actually for, the pattern-action structure, and your first real one-liners

AWK is a programming language built around one specific idea: process text **line by line**, splitting each line into fields automatically, and run an action whenever a pattern matches. Where SED (the companion course to this one) excels at find-and-replace text transformation, AWK excels at anything involving structured, column-based data — log files, CSVs, command output — anywhere "this is field 3 of this line" is a meaningful thing to say.

The Pattern-Action Structure

Every AWK program, no matter how complex, is built from the same fundamental unit: a pattern paired with an action. AWK reads input one line at a time, checks the pattern against the current line, and runs the action if it matches.

```
pattern { action }
↑ when should this run?    ↑ what should it do?
```

```
$ # Pattern: lines containing "ERROR" — Action: print the whole line
$ awk '/ERROR/ { print }' app.log
```

Either half can be omitted, with a specific meaning each time:

No pattern, just { action }

Runs the action on **every** line — no filtering at all.

Just a pattern, no { }

Defaults to printing the entire matching line — the most common AWK one-liner shape, equivalent to `grep` for that pattern.

```
$ # Just a pattern — print every matching line (like grep)
$ awk '/ERROR/' app.log

$ # Just an action — runs for every single line, no filtering
$ awk '{ print "Line: " $0 }' app.log
```

Your First Real Programs

```
$ # Print the whole line, exactly like cat
$ awk '{ print }' file.txt

$ # Print only lines longer than 80 characters
$ awk 'length > 80' file.txt

$ # Print lines matching a regex pattern (more on this in Chapter 3)
$ awk '/^#/ { print "comment: " $0 }' config.conf
```

Multiple Pattern-Action Pairs

A single AWK program can have several pattern-action pairs — each line of input is checked against every pattern in order, and every matching action runs (not just the first match).

```
$ awk '
/ERROR/ { print "Found error: " $0 }
/WARN/ { print "Found warning: " $0 }
' app.log
```

AWK vs grep vs sed — When to Reach for Each

🔍 grep

Find lines matching a pattern. Nothing more — no field access, no transformation.

✂️ sed

Transform text in place — substitution, deletion, insertion. Operates on the whole line as text, not structured fields.

📊 AWK

Everything grep and sed can do, plus genuine field-based processing, variables, arithmetic, and control flow — a real (if small) programming language.

IT'S COMMON, AND OFTEN CORRECT, TO PIPE ALL THREE TOGETHER

A realistic command-line workflow might use `grep` to narrow down which lines matter, pipe into `awk` to extract specific fields or compute something, and finish with `sed` for a final text cleanup —

three small, composable tools rather than one tool trying to do everything. Chapter 8 of this course returns to exactly this kind of combined pipeline.

Running AWK — Three Ways

```
$ # Inline, single-quoted (the most common way, used throughout this course)
$ awk '{ print $1 }' file.txt

$ # From a script file, with -f
$ awk -f myscript.awk file.txt

$ # Reading from a pipe instead of a file
$ ps aux | awk '{ print $1 }'
```

ALWAYS SINGLE-QUOTE AWK PROGRAMS IN THE SHELL

AWK programs use `$1`, `$2`, etc. for field references (Chapter 2) — inside double quotes, the shell would try to expand those as its own variables before AWK ever sees them. Single quotes prevent any shell expansion, letting AWK interpret the program exactly as written.

Command Reference

FORM	WHAT IT DOES
`awk 'pattern { action }' file`	Run action on lines matching pattern
`awk '{ action }' file`	Run action on every line, no filtering
`awk 'pattern' file`	Print every matching line (default action)
`awk -f script.awk file`	Run a program from a script file

CHAPTER 1 QUICK REFERENCE

- **pattern { action }** — the fundamental unit of every AWK program
- **No pattern** = runs on every line; **no action** = defaults to printing the matched line
- **Multiple pattern-action pairs** are checked independently against every line — all matches run, not just the first

- **AWK vs grep vs sed:** grep finds, sed transforms text, AWK does structured field processing with real programming constructs
- **Always single-quote** AWK programs in the shell — prevents the shell from expanding $1, $2, etc. itself
- **Next chapter:** fields and records — $0, $1, NF, NR, and field separators

CHAPTER 2: FIELDS AND RECORDS

CHAPTER 2

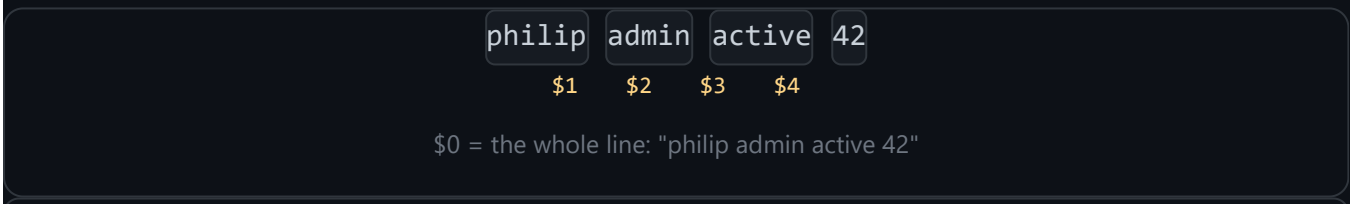
Fields and Records

$0, $1, NF, NR, and the field separators that control how AWK splits each line

This chapter is the real foundation of why AWK exists at all — automatic field splitting. Every line AWK reads (a "record") gets split into pieces (its "fields") without you writing any splitting logic yourself. Understanding exactly how that splitting works, and how to control it, unlocks almost everything else in this course.

Fields — $0, $1, $2...

By default, AWK splits each line on whitespace (spaces and/or tabs, any amount) into numbered fields, accessible as `$1`, `$2`, and so on. `$0` is special — it always refers to the entire current line, unsplit.




```
$ # Input line: "philip admin active 42"
$ echo "philip admin active 42" | awk '{ print $1 }'
philip

$ echo "philip admin active 42" | awk '{ print $2, $4 }'
admin 42

$ # A field beyond the actual number of fields is just empty, not an error
$ echo "a b" | awk '{ print $5 }'
(empty line)
```

The Built-In Variables That Describe the Current Line

NF	NR

Number of Fields — how many fields the current line was split into. `$NF` conveniently refers to the *last* field, whatever its number happens to be.

Number of Records — which line number AWK is currently processing, starting from 1. Persists across the entire input, even across multiple files.

FNR

File Number of Records — like NR, but resets to 1 for each new file when processing multiple files. NR keeps climbing; FNR restarts.

```
$ # NF — print how many fields each line has
$ awk '{ print NF }' file.txt

$ # $NF — print the LAST field of each line, regardless of how many there are
$ awk '{ print $NF }' file.txt

$ # NR — number every line, like cat -n
$ awk '{ print NR, $0 }' file.txt

$ # Only print line 5 specifically
$ awk 'NR==5' file.txt
```

$NF IS GENUINELY USEFUL, NOT JUST A CURIOSITY

Log lines and command output often have a variable number of fields, with the value you actually want always being the *last* one regardless of how many fields came before it — `$NF` handles that without needing to know NF's actual value in advance.

Field Separators — FS and OFS

Default whitespace splitting works for a lot of text, but plenty of real data is comma-separated, colon-separated, or otherwise structured differently. `FS` (Field Separator) controls how AWK splits the input; `OFS` (Output Field Separator) controls what gets placed between fields when printing them back out together.

```
$ # CSV-style input — set FS with -F, or inside a BEGIN block
$ awk -F',' '{ print $2 }' data.csv

$ # Colon-separated — classic /etc/passwd format
$ awk -F':' '{ print $1, $3 }' /etc/passwd
```

```
$ # Changing OFS affects how fields are joined when you print several together
$ awk -F',' 'BEGIN{OFS="|"} { print $1, $2 }' data.csv
name|value
```

OFS ONLY TAKES EFFECT WHEN YOU REBUILD $0 OR PRINT MULTIPLE FIELDS WITH COMMAS

Just setting OFS doesn't retroactively change the original line — it only applies when AWK actually joins fields together, either via `print $1, $2` (comma-separated print arguments) or by reassigning any field (which forces AWK to rebuild $0 using OFS). Printing `$0` directly without modifying any field still shows the original separators untouched.

BEGIN and END blocks — running code outside the per-line loop

Setting `OFS` above used a `BEGIN` block — code that runs once, before any input is read, rather than once per line. There's a matching `END` block that runs once after all input has been processed. Both get full coverage in Chapter 3, but they're worth introducing here since FS/OFS setup is one of their most common uses.

Command Reference

REFERENCE	WHAT IT GIVES YOU
`$0`	The entire current line, unsplit
`$1, $2, ...`	Individual fields, numbered from 1
`$NF`	The last field, regardless of how many fields exist
`NF`	The number of fields in the current line
`NR`	The current line number, across all input
`FNR`	The current line number, reset per file
`-F','`	Sets the input field separator from the command line
`BEGIN{FS=","; OFS="|"}`	Sets separators inside the program itself

CHAPTER 2 QUICK REFERENCE

- **$0** — the whole line; **$1, $2...** — individual fields, split by whitespace by default
- **$NF** — always the last field, whatever NF happens to be that line

- **NF** — field count; **NR** — running line count across all input; **FNR** — line count, resets per file
- **FS** — controls input splitting (set via -F or BEGIN); **OFS** — controls output joining
- **OFS only applies when fields are actually rejoined** — printing $0 unmodified shows the original text untouched
- **Next chapter:** patterns and conditions — regex patterns, ranges, and the BEGIN/END blocks introduced here in full

CHAPTER 3

Patterns and Conditions

Regex patterns, comparison expressions, ranges, and the BEGIN/END blocks introduced in Chapter 2

Chapter 1 introduced `/ERROR/` as a pattern without explaining what makes something a valid pattern at all. This chapter covers the full range of pattern types AWK supports — regex, comparisons, ranges — plus the two special patterns, `BEGIN` and `END`, that don't match lines at all but control setup and teardown.

The Pattern Types

🔤 Regex pattern

A regular expression between slashes — matches if it's found anywhere in the current line (or a specific field, if explicitly targeted).

```
/^ERROR/ { print }
```

🔢 Comparison expression

Any expression that evaluates to true/false — comparing fields, NR, NF, or computed values.

```
NF > 5 { print }
```

↔️ Range pattern

Two patterns separated by a comma — matches every line from the first match through the second match, inclusive.

```
/START/,/END/ { print }
```

🎬 BEGIN / END

Not tied to any line at all — BEGIN runs once before any input is read; END runs once after all input is processed.

```
BEGIN { print "Starting..." }
```

Regex Patterns in Depth

```
$ # Anchors work exactly as in grep/sed regex
$ awk '/^ERROR/' app.log # lines starting with ERROR
$ awk '/\.log$/' filelist.txt # lines ending in .log

$ # Matching against a SPECIFIC field, not the whole line — use ~
```

```
$ awk '$3 ~ /^admin/' users.txt

$ # Negated match with !~
$ awk '$3 !~ /^admin/' users.txt
```

/PATTERN/ ALONE CHECKS THE WHOLE LINE; FIELD ~ /PATTERN/ CHECKS ONE SPECIFIC FIELD

A bare `/ERROR/` pattern matches if ERROR appears *anywhere* in the line, including in a field you don't actually care about. `$3 ~ /ERROR/` is far more precise — it only matches if field 3 specifically contains it, ignoring the rest of the line entirely.

Comparison Expressions

PATTERN	MATCHES LINES WHERE...
`NF > 3`	The line has more than 3 fields
`$1 == "admin"`	Field 1 is exactly "admin"
`$4 >= 100`	Field 4, treated as a number, is 100 or more
`NR % 2 == 0`	The line number is even (every other line)
`$2 != ""`	Field 2 is not empty

```
$ # Print only lines where field 4 (e.g. a price) exceeds 100
$ awk '$4 > 100' sales.csv

$ # Combine conditions with && (and) / || (or)
$ awk '$4 > 100 && $2 == "active"' sales.csv
```

Range Patterns

A range pattern matches every line starting from where the first sub-pattern matches, through the line where the second sub-pattern matches — genuinely useful for extracting a section out of a larger structured file.

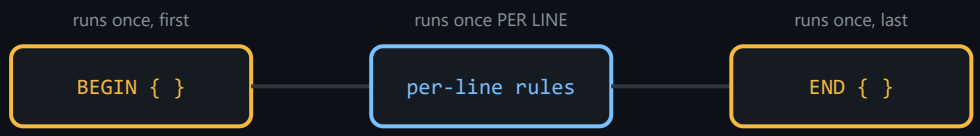
```
$ # Extract everything between two markers, inclusive
$ awk '/^\[database\]/,/^\[\/database\]/' config.ini
```

A RANGE PATTERN ALWAYS COMPLETES ONCE STARTED — EVEN IF THE END MARKER NEVER APPEARS

> If the second pattern in a range never matches for the rest of the file, AWK keeps the range "open" and matches every remaining line through to the end of input. Worth double-checking the end marker genuinely exists if a range pattern seems to be matching far more than expected.

BEGIN and END — Setup and Teardown

Chapter 2 used a `BEGIN` block to set `FS` / `OFS` before any line was processed. The full picture: AWK's execution has three distinct phases, and BEGIN/END are how you hook into the first and last.




```
BEGIN and END are the only two phases NOT tied to a specific line — everything else runs once per line of input
```

```
$ # A simple line counter using BEGIN and END together
$ awk '
BEGIN { print "Starting scan..." }
/ERROR/ { count++ }
END { print "Found " count " errors" }
' app.log
```

That `count` variable persists across every line because AWK variables aren't scoped to a single pattern-action block — they live for the entire run, which is exactly what makes accumulating totals across BEGIN → per-line → END possible. Chapter 6 covers variables and arrays in full depth.

CHAPTER 3 QUICK REFERENCE

- **/regex/** — matches if found anywhere in the line; **field ~ /regex/** — matches only within that field
- **!~** — negated match; combine conditions with **&&** / **||**
- **Comparison patterns** — NF, NR, $N compared with ==, !=, >, <, etc.
- **/start/,/end/** — range pattern; matches from the first match through the second, inclusive
- **A range with no matching end** stays open through the rest of the input — verify the end marker exists
- **BEGIN { }** — runs once before any line; **END { }** — runs once after all lines

- **Variables persist across the whole run** — the foundation of counting/accumulating patterns covered fully in Chapter 6
- **Next chapter:** built-in variables and functions — string functions, math functions

CHAPTER 4

Built-In Variables and Functions

String functions and math functions — the toolkit that turns simple field access into genuine text processing

Chapters 2 and 3 covered AWK's built-in variables that describe the current line — NF, NR, the field separators. This chapter covers the built-in *functions* available inside any action block: string manipulation and arithmetic that don't require any special setup, just calling them like any function call.

String Functions

FUNCTION	WHAT IT DOES
`length(s)`	Number of characters in string s (or the current line, $0, if no argument given)
`substr(s, start, len)`	Extracts a substring of s, starting at position start, len characters long
`toupper(s) / tolower(s)`	Returns s converted to upper/lowercase
`split(s, arr, sep)`	Splits string s into array arr, using sep as the delimiter — covered fully in Chapter 6's arrays
`index(s, target)`	Position where target first appears within s (0 if not found)
`gsub(regex, replacement, s)`	Replace every match of regex within s — AWK's equivalent of SED's global substitution
`sub(regex, replacement, s)`	Replace only the FIRST match of regex within s

```
$ # length — works on $0 by default, or any string
$ awk '{ print length }' file.txt # length of each line
$ awk '{ print length($1) }' file.txt # length of field 1 specifically

$ # substr — extract part of a string
$ awk '{ print substr($0, 1, 10) }' file.txt # first 10 characters
```

```
$ # toupper/tolower — case-insensitive comparisons
$ awk 'toupper($1) == "ADMIN"' users.txt

$ # gsub — replace every occurrence, modifying the variable in place
$ awk '{ gsub(/foo/, "bar"); print }' file.txt
```

GSUB AND SUB MODIFY THEIR TARGET IN PLACE AND RETURN A COUNT

Unlike most functions covered in this chapter, `gsub` / `sub` don't just return a new string — they directly modify the variable passed as the third argument (often `$0` or a specific field), and separately return the number of substitutions made. `gsub(/foo/, "bar", $2)` changes field 2 directly; calling it on $0 or a field also forces AWK to rebuild the line using OFS, exactly as covered in Chapter 2.

Math Functions

FUNCTION	WHAT IT DOES
`int(x)`	Truncates x to an integer (toward zero, not rounding)
`sqrt(x)`	Square root of x
`sin(x) / cos(x)`	Trigonometric functions, x in radians
`rand()`	A pseudo-random number between 0 and 1
`srand(x)`	Seeds the random number generator — call once, typically in BEGIN

```
$ # Sum a numeric column and print the total
$ awk '{ total += $3 } END { print total }' sales.csv

$ # Average — needs a count alongside the running total
$ awk '{ total += $3; count++ } END { print total/count }' sales.csv

$ # int() to drop decimal places from that average
$ awk '{ total += $3; count++ } END { print int(total/count) }' sales.csv
```

AWK TREATS STRINGS AND NUMBERS INTERCHANGEABLY, AUTOMATICALLY

A field like `$3` is just text as far as AWK's parser is concerned, but the moment it's used in a numeric context (`+=`, comparison with `>`, etc.), AWK automatically converts it. This "automatic type coercion" is convenient most of the time, but worth being aware of — comparing `$1 == "007"` as a string behaves differently than `$1 == 7` as a number, since "007" and 7 are equal numerically but not as text.

Combining String and Math Functions

```
$ # Truncate long lines to a fixed width, showing original length
$ awk '{ print substr($0, 1, 50), "(" length($0) " chars total)" }' file.txt

$ # Normalise case AND extract a substring together
$ awk '{ print substr(tolower($1), 1, 3) }' names.txt
```

CHAPTER 4 QUICK REFERENCE

- **length(s) / substr(s, start, len)** — size and extraction
- **toupper(s) / tolower(s)** — case conversion, useful for case-insensitive comparisons
- **index(s, target)** — find a substring's position; 0 if not found
- **gsub/sub(regex, replacement, target)** — modify target in place, return count of replacements made
- **int(x), sqrt(x)** and friends — standard math operations
- **Running totals:** total += $N inside the per-line action, read the result in END
- **AWK auto-converts between strings and numbers** based on context — be careful comparing "007" vs 7
- **Next chapter:** printf and formatted output

CHAPTER 5

printf and Formatted Output

Going beyond plain print when output needs to be aligned, padded, or precisely formatted

Every example so far has used `print`, which is simple but offers no control over spacing or formatting — fields just get separated by OFS and that's it. `printf` gives full control: column alignment, fixed decimal places, padded numbers — genuinely useful the moment output needs to look like a readable table rather than just raw values.

The Basic Syntax

```
$ awk '{ printf "%s is %d years old\n", $1, $2 }' people.txt
```

A format string with placeholders (`%s`, `%d`, etc.), followed by a comma-separated list of values to fill those placeholders, in order. Unlike `print`, `printf` does **not** add a newline automatically — the `\n` has to be written explicitly, every time.

FORGETTING \N IS THE SINGLE MOST COMMON PRINTF MISTAKE

Without it, every line of output runs together with no break at all — a wall of concatenated text instead of one line per record. `print` spoiled this expectation by adding a newline for you automatically; `printf` never does.

Format Specifiers

SPECIFIER	FORMATS AS
`%s`	String, as-is
`%d`	Integer (decimal)
`%f`	Floating-point number, default 6 decimal places
`%.2f`	Floating-point, exactly 2 decimal places — for currency, percentages
`%5d`	Integer, right-padded to a minimum width of 5 characters

SPECIFIER	FORMATS AS
`%-10s`	String, LEFT-aligned, padded to a minimum width of 10 characters
`%%`	A literal percent sign — needed because % alone starts a specifier

Building an Aligned Table

This is where printf genuinely earns its place over plain print — turning ragged, inconsistently-spaced output into a clean, column-aligned table.

```
$ awk '{ printf "%-10s %5d %8.2f\n", $1, $2, $3 }' sales.txt
```

```
philip          42   199.50
admin            7    49.99
guest          156  1024.00
```

`%-10s` left-aligns the name within 10 characters; `%5d` right-aligns the integer within 5; `%8.2f` right-aligns the decimal within 8 characters, always showing exactly 2 decimal places — regardless of how differently-sized the original values were.

> **A HEADER ROW WITH THE SAME WIDTHS MAKES A GENUINELY READABLE REPORT**
>
> Adding a `BEGIN` block (Chapter 3) with a matching `printf` for column headers, using the same width specifiers, produces output that looks like a real formatted table rather than a raw data dump — a small addition that makes scripted reports far more usable.

```
$ awk '
BEGIN { printf "%-10s %5s %8s\n", "NAME", "AGE", "TOTAL" }
{ printf "%-10s %5d %8.2f\n", $1, $2, $3 }
' sales.txt
```

```
NAME          AGE    TOTAL
philip          42   199.50
admin            7    49.99
guest          156  1024.00
```

printf Inside sprintf — Building Strings Without Printing

`sprintf` behaves identically to `printf` but returns the formatted result as a string instead of printing it — useful when the formatted text needs to be used in further processing rather than output immediately.

```
$ awk '{ label = sprintf("%-10s (%d)", $1, $2); print label }' file.txt
```

CHAPTER 5 QUICK REFERENCE

- **printf "format", values** — no automatic newline, unlike print; \n must be explicit
- **%s, %d, %f** — string, integer, floating-point; **%.2f** — fixed decimal places
- **%5d** — right-aligned, min width 5; **%-10s** — left-aligned, min width 10
- **%%** — a literal percent sign
- **Combine width specifiers** across fields to build genuinely aligned table output
- **sprintf** — same formatting, returns a string instead of printing it
- **Next chapter:** arrays in AWK — associative arrays, counting, grouping

CHAPTER 6

Arrays in AWK

Associative arrays, and the counting/grouping/deduplication patterns that make them genuinely powerful

This is the chapter where AWK stops feeling like a text filter and starts feeling like a real data-processing tool. AWK arrays are **associative** — indexed by arbitrary strings, not just sequential numbers — which makes them perfect for exactly the kind of "group by" and "count occurrences of" operations that come up constantly with real-world data.

Associative Arrays — The Basics

No declaration needed — an array springs into existence the first time you assign to any index of it.

```
$ awk 'BEGIN {
fruit["apple"] = 5
fruit["banana"] = 3
print fruit["apple"]
}'
5
```

```
fruit["apple"]  → 5
fruit["banana"]  → 3
```

Pattern 1 — Counting Occurrences

The single most common array pattern in all of AWK: count how many times each distinct value appears.

```
$ # Count how many log lines came from each status code
$ awk '{ counts[$9]++ } END { for (code in counts) print code, counts[code] }' access.log
200 1543
404 87
500 12
```

`counts[$9]++` works even for a status code that's never been seen before — AWK treats an unset array index as 0 in a numeric context, so the first increment correctly produces 1. No explicit "if this key doesn't exist yet, initialise it" check is needed.

Pattern 2 — Grouping Values

Sometimes the goal isn't a count, but accumulating a running total or list per group — same fundamental pattern, different operation per match.

```
$ # Sum sales total per category (field 1 = category, field 3 = amount)
$ awk '{ totals[$1] += $3 } END { for (cat in totals) printf "%-10s %.2f\n", cat,
totals[cat] }' sales.csv
books 245.50
electronics 1830.00
```

Pattern 3 — Deduplication

Using array keys as a "seen before" check is a clean, idiomatic way to print only unique values, without needing to sort first or compare against previous lines manually.

```
$ # Print each unique value in field 1 exactly once, first-seen order preserved
$ awk '!seen[$1]++' file.txt
```

This is a famously compact AWK idiom worth understanding fully: `seen[$1]++` increments the count for this value and returns its *previous* value (0 the first time, then 1, 2...). `!0` is true, so the first occurrence of any value passes the implicit pattern (and prints, per Chapter 1's "pattern with no action defaults to print"); every subsequent occurrence evaluates `!1`, `!2`, etc. — all false, so they're filtered out.

> **!SEEN[$1]++ IS GENUINELY ONE OF THE MOST QUOTED AWK ONE-LINERS IN EXISTENCE**
>
> It accomplishes in a few characters what would otherwise need an explicit if/else and a separate counting step — exactly the kind of expressive compactness AWK is known for, once the underlying logic actually makes sense rather than being memorised as magic.

Iterating Over an Array — for...in

```
$ for (key in array) { print key, array[key] }
```

FOR...IN DOES NOT GUARANTEE ANY PARTICULAR ORDER

Unlike iterating over a numerically indexed list in most languages, AWK's `for (key in array)` visits keys in an implementation-defined order — not insertion order, not alphabetical, not numerical. If a specific output order matters (alphabetical, by count descending), pipe the AWK output into `sort` afterward rather than trying to control ordering inside AWK itself.

```
$ # Get counts, then sort the OUTPUT by count descending using the sort command
$ awk '{ counts[$9]++ } END { for (c in counts) print counts[c], c }' access.log | sort -rn
```

Checking Whether a Key Exists — Without Creating It

```
$ # The wrong way — this CREATES the key just by referencing it
$ if (arr["maybe"] == "") { ... }

$ # The correct way — (key in array) checks existence without creating it
$ if ("maybe" in arr) { print "exists" }
```

SIMPLY REFERENCING ARR[KEY] CREATES THAT KEY AS A SIDE EFFECT

This catches people out constantly: just checking `arr["maybe"]` in a condition (rather than using the dedicated `in` operator) silently creates an empty entry for "maybe" if it didn't already exist — which can quietly inflate a later `for...in` loop or array length count. Always use `(key in array)` for a genuine existence check.

Command Reference

SYNTAX	WHAT IT DOES
`arr[key] = value`	Assign a value, creating the key if it doesn't exist
`arr[key]++`	Increment a counter, treating a missing key as 0
`for (key in arr)`	Iterate over all keys, in unspecified order
`(key in arr)`	Check existence WITHOUT creating the key as a side effect
`delete arr[key]`	Remove a specific key from the array

CHAPTER 6 QUICK REFERENCE

- **Associative arrays** — indexed by arbitrary strings, created on first assignment, no declaration needed
- **Counting:** counts[key]++ — missing keys treated as 0 automatically
- **Grouping/totals:** totals[key] += value, read in END
- **Deduplication:** !seen[$1]++ — a compact, genuinely useful idiom once understood
- **for (key in arr)** — order is NOT guaranteed; pipe to sort if order matters
- **(key in arr)** — checks existence WITHOUT creating the key; referencing arr[key] directly DOES create it
- **Next chapter:** control flow — if/else, for and while loops in AWK

CHAPTER 7

Control Flow

if/else, for loops, and while loops — the constructs that make AWK a genuine programming language, not just a pattern matcher

Everything so far has relied on AWK's implicit per-line loop and pattern matching. This chapter covers the control flow constructs available *inside* an action block — letting an action make decisions and repeat work on its own, independent of the line-by-line structure covering the rest of this course.

if / else

```
$ awk '{ if ($3 > 100) { print $1, "high value" } else if ($3 > 50) { print $1, "medium value" } else { print $1, "low value" } }' sales.csv
```

Behaves exactly as in most C-family languages — braces are optional for a single statement, but using them consistently (as above) avoids a common class of mistake when adding a second statement to a branch later and forgetting to add braces at the same time.

The Ternary Operator — A Compact if/else for Expressions

```
$ awk '{ status = ($3 > 100) ? "high" : "low"; print $1, status }' sales.csv
```

Useful for short, single-value decisions inline — generally not a good substitute for a full if/else once more than one branch of logic is genuinely needed.

for Loops

AWK's C-style `for` loop is the standard choice for counting a fixed number of times — and combines naturally with field access, since fields are just numbered.

```
$ # Print every field on its own line, regardless of how many fields exist
$ awk '{ for (i = 1; i <= NF; i++) print $i }' file.txt
```

```
$ # Sum all fields on each line
$ awk '{ sum = 0; for (i = 1; i <= NF; i++) sum += $i; print sum }' numbers.txt
```

> **FOR (I = 1; I <= NF; I++) IS THE STANDARD IDIOM FOR "DO SOMETHING TO EVERY FIELD"**
>
> Because NF (Chapter 2) gives the exact field count for the current line, this loop pattern correctly handles lines with any number of fields without hardcoding a number — genuinely one of the most reused patterns in everyday AWK scripts.

for...in for arrays — a different for loop entirely

Chapter 6 already covered `for (key in array)` — note this is a completely different loop form from the C-style for loop above, despite sharing the keyword. The C-style form counts; the `in` form iterates over array keys, in unspecified order.

while Loops

```
$ # Equivalent to the for-loop field example above, written with while
$ awk '{ i = 1 while (i <= NF) { print $i i++ } }' file.txt
```

AWK also supports `do...while`, which runs the loop body at least once before checking the condition — less commonly needed than `while`, but available for the rare case where "run once, then keep going while true" is genuinely the right shape.

break and continue

break

Exits the innermost loop immediately, skipping any remaining iterations entirely.

continue

Skips the rest of the current iteration only, then continues with the next one — the loop keeps running.

next

Specific to AWK's outer per-line loop (not the for/while loops above) — skips the rest of the current LINE's processing entirely, moving straight to the next line of input.

```
$ # Skip blank lines entirely — next applies to the outer per-line loop
$ awk '{ if (NF == 0) next; print }' file.txt

$ # break out of a for loop once a specific field is found
$ awk '{ for (i = 1; i <= NF; i++) { if ($i == "STOP") break print $i } }' file.txt
```

NEXT SKIPS THE REST OF THE CURRENT LINE'S ACTION, NOT THE CURRENT LOOP ITERATION

A common confusion: `next` isn't a loop control statement at all — it operates on AWK's implicit outer loop over input lines (Chapter 1's per-line execution model), completely separate from any for/while loop written inside an action. Using `next` inside a for loop skips the rest of the entire line's processing, not just the rest of that loop.

Command Reference

CONSTRUCT	USE
`if / else if / else`	Conditional branching
`cond ? a : b`	Compact inline conditional expression
`for (i=1; i<=N; i++)`	C-style counting loop — pairs naturally with NF
`for (key in array)`	Iterate array keys (Chapter 6) — different from the loop above despite shared keyword
`while (cond)`	Loop while a condition holds
`break / continue`	Exit or skip within the current for/while loop
`next`	Skip to the next INPUT LINE entirely — not a loop construct

CHAPTER 7 QUICK REFERENCE

- **if/else if/else** — standard branching; use braces consistently to avoid future mistakes
- **cond ? a : b** — ternary, good for short single-value decisions only
- **for (i=1; i<=NF; i++)** — the standard idiom for "do this to every field"
- **for (key in array)** is a different construct entirely from the C-style for loop, despite the shared keyword
- **while / do...while** — condition-driven repetition

- **break/continue** — control the current for/while loop only
- **next** — skips the rest of the current LINE, not the current loop — operates on AWK's outer per-line execution
- **Next chapter (final):** real-world one-liners and scripts, combining AWK with grep and sed in pipelines

CHAPTER 8 · FINAL CHAPTER · CAPSTONE

Real-World One-Liners and Scripts

Log analysis, CSV processing, and combining AWK with grep and sed in realistic command-line pipelines

Every previous chapter introduced one mechanism at a time. This final chapter is entirely practical — genuine one-liners for tasks that come up constantly, and a look at how AWK fits alongside grep and sed (introduced back in Chapter 1) in real combined pipelines rather than in isolation.

Log Analysis

```
$ awk '{ print $1 }' access.log | sort | uniq -c | sort -rn | head -10
```

Top 10 most frequent IP addresses in an Apache/nginx access log. Extract field 1 (IP), count occurrences with sort+uniq -c, sort descending, take the top 10. AWK does the extraction; the rest is a classic Unix pipeline.

```
$ awk '$9 >= 500' access.log
```

Lines where the HTTP status code (field 9 in standard Apache log format) is 500 or higher — every server error, instantly filtered from a huge log file.

```
$ awk '{ sizes[$9] += $10 } END { for (s in sizes) print s, sizes[s] }' access.log
```

Total bytes transferred (field 10), grouped by status code (field 9) — combines Chapter 6's grouping pattern directly on real log data.

CSV Processing

```
$ awk -F',' 'NR > 1 { sum += $3 } END { print "Total:", sum }' sales.csv
```

`NR > 1` skips the header row (Chapter 3's comparison pattern) — a near-universal first line in any CSV-processing AWK command.

```
$ awk -F',' '{ print $2 }' data.csv | sort -u
```

Every distinct value in column 2, alphabetically — using the external sort -u rather than Chapter 6's !seen[] idiom, when sorted order matters more than first-seen order.

```
$ awk -F',' '{ print $1","$3","$2 }' data.csv
```

Reorder CSV columns — print fields back out in a different order than they appeared, with explicit comma separators since the output isn't using OFS automatically here.

System Administration

```
$ ps aux | awk '{ print $4, $11 }' | sort -rn | head -5
```

Top 5 processes by memory percentage (field 4) alongside their command (field 11) — real-time system inspection using a tool already on every Linux/Mac system.

```
$ df -h | awk '$5+0 > 80 { print $1, $5 }'
```

Disk partitions over 80% full. `$5+0` forces the percentage string ("85%") into a number for comparison — the +0 trick strips the % since AWK's numeric conversion stops at the first non-numeric character.

Combining grep, AWK, and SED in One Pipeline

Chapter 1 introduced the principle: each tool does the part it's genuinely best at. A realistic combined pipeline looks like this:

```
Each tool handles exactly the part it's built for — none of the three try to do everything alone
```

```
$ grep "ERROR" app.log | awk -F'|' '{ print $2, $4 }' | sed 's/^[ \t]*//'
```

grep narrows to error lines only; awk extracts just the timestamp and message fields from a pipe-delimited format; sed strips any leading whitespace from the final output. Each command stays simple because it only does one job.

```
$ awk '/ERROR/,/RESOLVED/' app.log | grep -v "DEBUG" | wc -l
```

AWK's range pattern (Chapter 3) extracts an incident's full log section; grep -v filters out noisy debug lines from within that section; wc -l counts what's left — a genuinely useful incident-analysis one-liner built from three simple pieces.

IF A "ONE-LINER" IS GETTING HARD TO READ, IT'S TIME FOR A SCRIPT FILE

Everything in this chapter fits comfortably on one line, but a real AWK script with several BEGIN/END blocks, multiple patterns, and real logic (Chapter 7) is more maintainable in a saved `.awk` file, run with `awk -f script.awk` (Chapter 1) — readability matters more than cleverness once a script is going to be reused or maintained by anyone, including future you.

CHAPTER 8 QUICK REFERENCE — AND COURSE WRAP-UP

- **Log analysis:** field extraction + sort/uniq -c for frequency counts; status-code filtering with comparison patterns
- **CSV:** NR > 1 to skip headers; sort -u or !seen[] for unique values
- **System admin:** ps/df piped into AWK for quick numeric filtering and sorting on live system data
- **+0 trick:** forces a string like "85%" into a number, stopping at the first non-numeric character
- **Combined pipelines:** grep narrows, AWK extracts/computes, sed does final text cleanup — each tool does one job
- **Move to a script file** once a one-liner starts being hard to read at a glance
- **Course recap:** intro & pattern-action (Ch1) → fields/records (Ch2) → patterns/BEGIN/END (Ch3) → functions (Ch4) → printf (Ch5) → arrays (Ch6) → control flow (Ch7) → real-world pipelines (Ch8)